Abstract

Major depression constitutes a serious challenge in personal and public health. Tens of millions of people each year suffer from depression and only a fraction receives adequate treatment. We explore the potential to use social media to detect and diagnose major depressive disorder in individuals.

However depression detection in social media can be considered a complex task, mainly due to the complicated nature of mental disorders. In recent years, this research area has started to evolve with the continuous increase in popularity of social media platforms that became an integral part of people's life. This close relationship between social media platforms and their users has made these platforms to reflect the users' personal life on many levels. In such an environment, researchers are presented with a wealth of information regarding one's life.

We create our own dataset by scraping tweets off various Twitter pages, and label them with the aid of polarity score generated from Textblob's python packages.Then we construct a deep learning model based on LSTM using pre-trained embeddings to generate predictions on this dataset. We find the model to be performing with 97% accuracy.

Introduction

Mental illness is a leading cause of disability worldwide. It is estimated that nearly 300 million people suffer from depression (World Health Organization, 2001). Reports on lifetime prevalence show high variance, with 3% reported in Japan to 17% in the US. In North America, the probability of having a major depressive episode within a one year period of time is 3–5% for males and 8–10% for females (Andrade et al., 2003). However, global provisions and services for identifying, supporting, and treating mental illness of this nature have been considered as insufficient (Detels, 2009). Although 87% of the world's governments offer some primary care health services to tackle mental illness, 30% do not have programs, and 28% have no budget specifically identified for mental health (Detels, 2009).

The rise of online social network provides unprecedented opportunities for solving problems in a wide variety of fields with information techniques . For example, traditional psychology research is based on questionnaires and academic interviews, but many psychologists are now turning their sights to web media. They try to analyze the data of social networks from the view of psychology. Undoubtedly, this discipline integration injects vigor into psychology, however, the supports from technical perspective are far from enough. Only some simple statistical tools are applied in such kind of research, and little attention is paid to design specific data mining methods. This project applies deep learning techniques to psychology, specifically the field of depression, to detect depressed users in social network services. The expansion of deep learning to psychology is of great technical and social significance. It is proved that the proposed model in this paper could effectively help for detecting depressed ones and preventing suicide in online social networks.

In this project, we work towards timely depression detection via harvesting social media. This work is non-trivial owning to the following challenges: 1) As far as we know, there is no public available large-scale benchmark datasets for depression research that are suitable to our study. 2) Users' behaviors on social media are multi-faceted. It is hard to characterize the users from discriminant perspectives and capture the relation across different modalities. 3) Although users' behaviors are rich and diverse, only a few are symptoms of depression, so the depressive-oriented features are sparse on social media and hard to be captured.
By using a wide set of twitter data and LSTM technique combined with features like level of user interaction on social media, we hope to create a robust model that is sensitive to the variations of depression on an individual basis.The findings of this project will be useful in predicting depression in individuals, even if they are unwilling to discuss their issues with a professional.

Our Work majorly consists of these steps. First, we first construct well-labeled depression and non-depression datasets on Twitter. Second, using a pre-trained model of word embeddings we analyze the context of the words in the dataset and project the depression inclination of the tweet. Third, a layered functional model is build using the depression inclination and other features extracted from the tweet like number of likes, replies and retweets. The model consists of embedding layer, LSTM layer and dense layer.This model is used to predict whether a tweet is from depressed person or not.

Literature Survey

Rich bodies of work on depression in psychiatry, psychology, medicine, and sociolinguistics describe efforts to identify and understand correlates of MDD in individuals. Cloninger et al., (2006) examined the role of personality traits in the vulnerability of individuals to a future episode of depression, through a longitudinal study. On the other hand, Rude et al., (2003) found support for the claim that negative processing biases, particularly (cognitive) biases in resolving ambiguous verbal information can predict subsequent depression. Robinson and Alloy, (2003) similarly observed that negative cognitive styles and stress-reactive rumination were predictive of the onset, number and duration of depressive episodes. Finally, Brown et al., (1986) found that lack of social support and lowered self-esteem are important factors linked to higher incidences of depression. These studies typically are based on surveys, relying on retrospective self-reports about mood and observations about health: a method that limits temporal granularity. That is, such assessments are designed to collect high-level summaries about experiences over long periods of time.

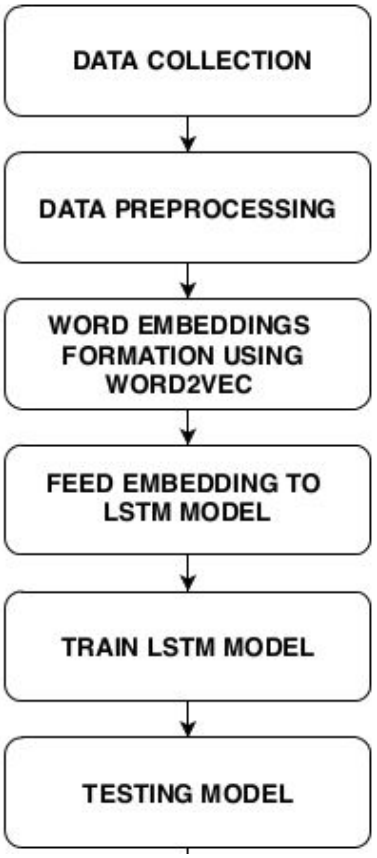
Moving to research on social media, over the last few years, there has been growing interest in using social media as a tool for public health, ranging from identifying the spread of flu symptoms (Sadilek et al., 2012), to building insights about diseases based on postings on Twitter (Paul & Dredze, 2011). However, research on harnessing social media for understanding behavioral health disorders is still in its infancy. Kotikalapudi et al., (2012) analyzed patterns of web activity of college students that could signal depression. Similarly, Moreno et al., (2011) demonstrated that status updates on Facebook could reveal symptoms of major depressive episodes.

In the context of Twitter, Park et al., (2012) found initial evidence that people post about their depression and even their treatment on social media. In other related work, De Choudhury et al., (2013) examined linguistic and emotional correlates for postnatal changes of new mothers, and built a statistical model to predict extreme postnatal behavioral changes using only prenatal observations.

Algorithm

Flowchart

The Algorithm includes the various steps which are shown in the flowchart below :



Steps Involved

1	**Collection of data**	A dataset of depressive tweets are collected from TWINT using keyword “depressive” and random tweets are collected the same way.
2	**Loading collected data**	The collected data is loaded using pandas which also converts it into a pandas dataframes for further processing
3	**Preprocessing the tweets**	Since tweets collected are not properly formatted, they are cleaned using NLTK. 1. First, http links are removed from the tweets 2. Second, fix faulty encoded words. 3. Third, expand contractions present in a tweet i.e phrases like “where’s” is changed to “where is”. 4. Remove stopwords from tweets such as “the”,”is” etc. 5. Using PorterStemmer perform stemming on tweets. For ex. “interesting” is changed to “interest”
4	**Preparation of no of likes, replies & retweets**	Prepare a list of no of likes replies and retweets to each tweet for both depressive as well as random tweets. Each element of the list corresponds to a tweet.
5	**Normalisation**	Normalise the list for both depressive and random tweets and then add no of like , replies and retweets to give a single number.
6	**Tokenizing Tweets**	Tokenizing is used to convert text into tokens. Each word is given a unique integer value.
7	**Word Embeddings**	The word embeddings are prepared for both depressive and random tweets using word2vec.
8	**Splitting data**	The dataset is split into training, testing and validation sets
9	**Creating Model**	Functional model is created using Keras. The layered structure of the form Embedding layer --> Convolutional Layer --> LSTM layer --> Dense Layer is created.
10	**Compiling the model**	The above layered model is compiled.
11	**Training**	The compiled model is trained for 10 epochs.
12	**Results**	The model is tested over the test data and accuracy of the model is determined.

Pseudocode

```
begin
dep_tweets := Load(deppressiveTweets.csv)    //loading tweets
ran_tweets := Load(randomTweets.csv)
clean_dep_tweets=Clean(dep_tweets)          //preprocessing tweets
clean_rand_tweets=Clean(ran_tweets)
tokenized_dep_tweets=Tokenize(cle/an_dep_tweets)     //tokenzing
tokenized_ran_tweets=Tokenize(clean_rand_tweets)
final_dep_tweets=formEmbeddings(tokenized_dep_tweets,Word2Vec)  //word
final_ran_tweets=formEmbeddings(tokenized_ran_tweets, Word2Vec)  //embeddings
labels_dep=[1]*size(final_dep_tweets)
labels_ran=[0]*size(final_ran_tweets)
train_data=0.6 * final_dep_tweets + 0.6 * final_ran_tweets  //Splitting data
train_labels=0.6 * labels_dep + 0.6 * labels_ran
test_data=0.2*final_dep_tweets + 0.2 * final_ran_tweets
val_labels=0.2 * labels_dep + 0.2 * labels_ran
val_data=0.2 * final_dep_data + 0.2 * final_ran_tweets
LSTM=lstm()                        //model creation and compiling
dense=Dense()
Model=add(LSTM)
Model=add(Dense)
Compile(Model)
model.fit([train_data,train_labels],[val_data,val_labels])    //model training
end
```

Prerequisite

Word2Vec

Word Embedding is a language modeling technique used for mapping words to vectors of real numbers. It represents words or phrases in vector space with several dimensions. Word embeddings can be generated using various methods like neural networks, co-occurrence matrix, kmin models, etc.

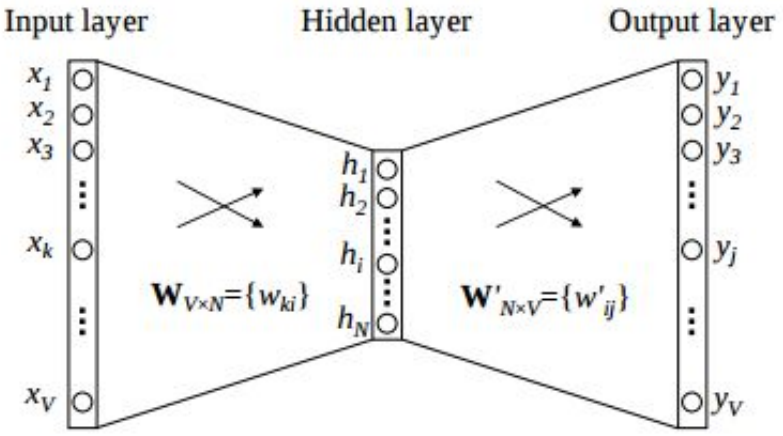
Word2Vec consists of models for generating word embedding. These models are shallow two layer neural networks having one input layer, one hidden layer and one output layer.

Word2Vec utilizes two architectures :

a). CBOW (Continuous Bag of Words)

CBOW model predicts the current word given context words within specific window. The input layer contains the context words and the output layer contains the current word. The hidden layer contains the number of dimensions in which we want to represent current word present at the output layer

Let us first see a diagrammatic representation of the CBOW model.

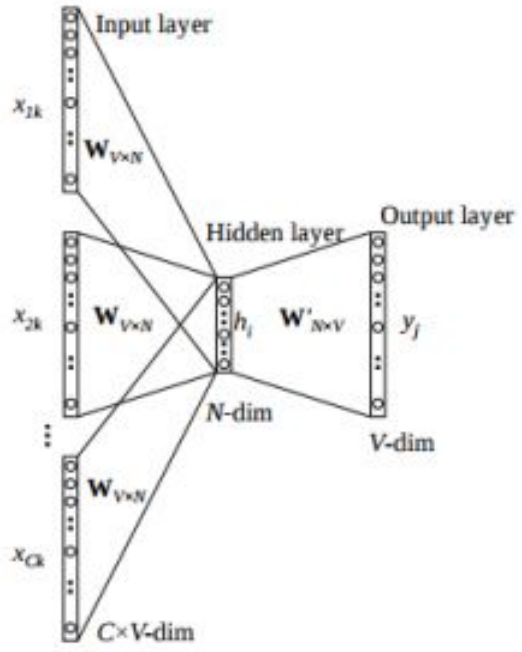


There are two sets of weights. one is between the input and the hidden layer and second between hidden and output layer. Input-Hidden layer matrix size =[V X N] , hidden-Output layer matrix size =[N X V] : Where N is the number of dimensions we choose to represent our word in. It is arbitary and a hyper-parameter for a Neural Network. Also, N is the number of neurons in the hidden layer. Here, N=4.

There is a no activation function between any layers.(More specifically, I am referring to linear activation). The input is multiplied by the input-hidden weights and called hidden activation. It is simply the corresponding row in the

input-hidden matrix copied.The hidden input gets multiplied by hidden- output weights and output is calculated.

The image below describes the architecture for multiple context words.

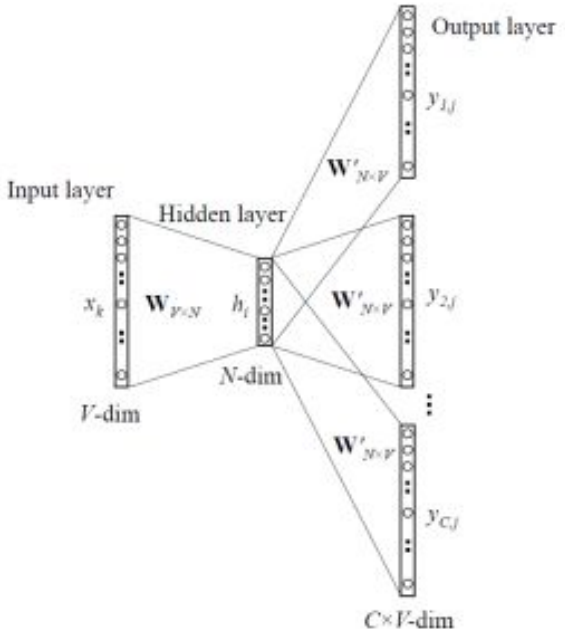


b). SKIP - GRAM Model

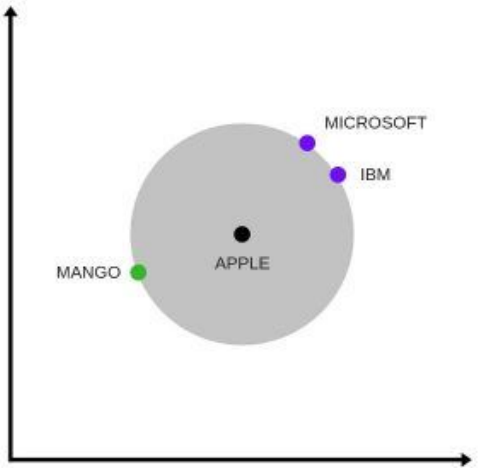
Skip – gram follows the same topology as of CBOW. It just flips CBOW's architecture on its head. The aim of skip-gram is to predict the context given a word. Let us take the same corpus that we built our CBOW model on. C="Hey, this is sample corpus using only one context word." Let us construct the training data.

The input vector for skip-gram is going to be similar to a 1-context CBOW model. Also, the calculations up to hidden layer activations are going to be the same. The difference will be in the target variable. Since we have defined a context window of 1 on both the sides, there will be "two" one hot encoded target variables and "two" corresponding outputs as can be seen by the blue section in the image.

The skip-gram architecture is shown below.



Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit. Skip-gram with negative sub-sampling outperforms every other method generally.



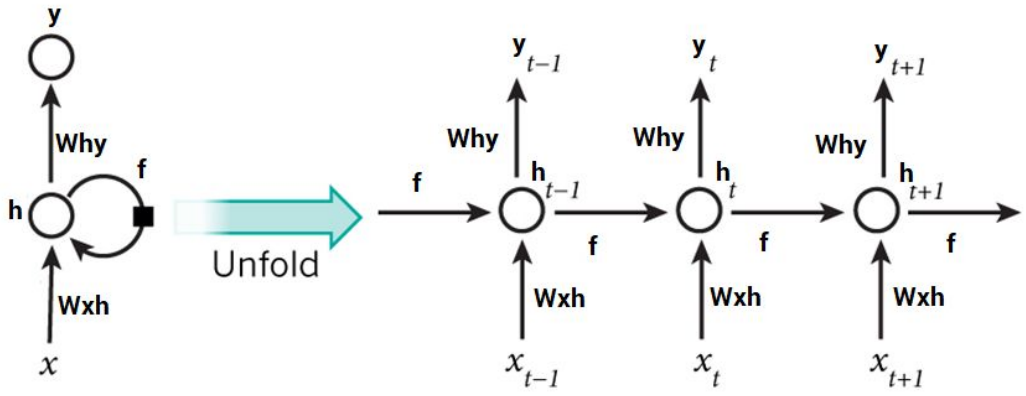
The above image is a t-SNE representation of word vectors in 2 dimension and you can see that two contexts of apple have been captured. One is a fruit and the other company.

LSTM

Lstm stands for Long Short-Term Memory. The model takes in an input and then outputs a single number representing the probability that the tweet indicates depression. The model takes in each input sentence, replace it with it's embeddings, then run the new embedding vector through a standard LSTM layer. Last but not least, the output of the LSTM layer is fed into a standard Dense model for prediction.

With the recent breakthroughs that have been happening in data science, it is found that for almost all of these sequence prediction problems, Long short Term Memory networks, a.k.a LSTMs have been observed as the most effective solution.

LSTMs have an edge over conventional feed-forward neural networks and RNN in many ways. This is because of their property of selectively remembering patterns for long durations of time. In the conventional feed-forward neural networks, all test cases are considered to be independent. A typical RNN looks like :



Limitations of RNN:

During the training of RNN, as the information goes in loop again and again which results in very large updates to neural network model weights. This is due to the accumulation of error gradients during an update and hence, results in an unstable network. At an extreme, values

of weights can become so large as to overflow and result in NaN values.The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1 or vanishing occurs if the values are less than 1.

RNNs can solve our purpose of sequence handling to a great extent but not entirely. Now RNNs are great when it comes to short contexts, but in order to be able to build a story and remember it, we need our models to be able to understand and remember the context behind the sequences, just like a human brain.

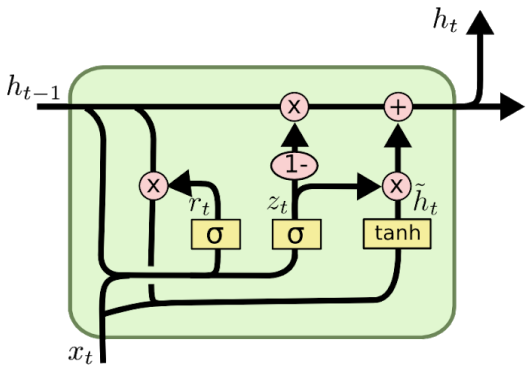
This is not possible with a simple RNN.This issue can be resolved by applying a slightly tweaked version of RNNs – the Long Short-Term Memory Networks. To overcome the vanishing gradient problem, we need a function whose second derivative can sustain for a long range before going to zero. *tanh* is a suitable function with the above property. As Sigmoid can output 0 or 1, it can be used to forget or remember the information.Information passes through many such LSTM units.

1. LSTM has a special architecture which enables it to forget the unnecessary information .The sigmoid layer takes the input X(t) and h(t-1) and decides which parts from old output should be removed (by outputting a 0). In our example, when the input is ‘He has a female friend Maria’, the gender of ‘David’ can be forgotten because the subject has changed to ‘Maria’. This gate is called forget gate f(t). The output of this gate is f(t)*c(t-1).

2. The next step is to decide and store information from the new input X(t) in the cell state. A Sigmoid layer decides which of the new information should be updated or ignored. A *tanh* layer creates a vector of all the possible values from the new input. These two are multiplied to update the new cell sate. This new memory is then added to old memory c(t-1) to give c(t). In our example, for the new input ‘ He has a female friend Maria’, the gender of Maria will be updated. When the input is ‘Maria works as a cook in a famous restaurant in New York whom he met recently in a school alumni meet’, the words like ‘famous’, ‘school alumni meet’ can be ignored and words like ‘cook, ‘restaurant’ and ‘New York’ will be updated.

3. Finally, we need to decide what we’re going to output. A sigmoid layer decides which parts of the cell state we are going to output. Then, we put the cell state through a *tanh* generating all the possible values and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to. In our example, we want to predict the blank word, our model knows that it is a noun related to ‘cook’ from its memory, it can easily answer it as ‘cooking’. Our model does not learn this answer from the immediate dependency, rather it learnt it from long term dependency.

Now let's get into the details of the architecture of LSTM network:



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Now, this is nowhere close to the simplified version which we saw before, but let me walk you through it. A typical LSTM network is comprised of different memory blocks called cells

(the rectangles that we see in the image). There are two states that are being transferred to the next cell; the cell state and the hidden state. The memory blocks are responsible for remembering things and manipulations to this memory is done through three major mechanisms, called gates.

Hence, LSTM outperforms the other models when we want our model to learn from long term dependencies. LSTM's ability to forget, remember and update the information pushes it one step ahead of RNNs.

Work Done

Dataset

The dataset consists of 30,000 tweets scraped from Twitter and each tweet is labeled with 1 or 0; 1 indicates that the tweet shows characteristics of a depressed person, 0 indicates that the user who posted the tweet does not suffer from depression. The train data set() and test data set() are randomly shuffled and selected from the above dataset.

Data Collection

The dataset is collected using an online Twitter scraping tool called Twint. All the scraped tweets are originally unlabeled. To collect potentially depressive tweets, keywords like “depression”,”anxiety”,”loneliness” etc is used in the query for scraping tweets. For the random tweets, no specific keywords is given. This isdone to obtain a wide range of tweets with other emotions as well so that the model is able to distinguish between emotions due to depression and other emotions.

After collection, the tweets are labeled manually. To aid the labeling, a script that uses the Textblob python package to calculate a polarity score for each tweet is made. Based on these scores, the script labels each tweet as 1 for depressed, 0 for nondepressed. After running this script, the tweets are manually filtered to check if the predictions made by the Textblob algorithm are reasonable, and labels are updated accordingly. An example ‘depressed’ tweet looks like this:

“It is tough, isn’t it? feeling like you’re not good enough, no matter how hard you try.”

Preprocessing

Cleaning the tweets

Removal of links, @, and hashtags and emojis
The tweets are cleaned by removing all the retweets, URL’s, @mentions, hashtags and all the non-alphanumeric characters including emojis. For example - “#sadness” is changed to “sadness”.

Correcting the encoding of the broken code using ftfy :
ftfy (fixes text for you) fixes Unicode that’s broken in various ways.
The goal of ftfy is to take in bad Unicode and output good Unicode, for use in your Unicode-aware code.

Expanding contracted text
The words ending with “n’t” are substituted with the complete word like - “can’t” is replaced with “cannot”. For repeating words whenever there are more than three letters together, the group of letters is replaced with a single letter. For example - word such as “hungryyy” is converted to “hungry”.

Removal of punctuations and stopwords
Stopwords such as “the”,”an”,”is”,”are” etc are removed from the data using NLTK stopwords Corpus.
Moreover, punctuations are also removed from the tweets.

Stemming
PorterStemmer is used for converting words to their root forms. For example – “strongest” is changed to “strong”.

Normalization

The number of replies for each tweet is extracted and stored in a numpy array. It is then normalized using the formula :

$$normal\_no\_replies[i] = normal\_no\_replies[i] \div max(\, no\_replies[0 : n - 1] \,)$$

Tokenization

Tokenizing is used to convert text into tokens. Each word is given a unique integer value. Then sequence of words is created using the tokens and the sequences are padded to make each one of uniform length.

Splitting data

The data is split into training, testing and validation sets with a ratio of 60:20:20.

	Depressive Tweets	**Normal Tweets**
Training	6,000	12,000
Validation	2,000	4,000
Testing	2,000	2,000
Total	10,000	20,000

Word Embedding Matrix

Word embedding models are fundamentally based on the unsupervised training of distributed representations, which can be used to solve supervised tasks. They are used to project words into a low-dimensional vector representation x_i, where $x_i \in R^W$ and W is the word weight embedding matrix.

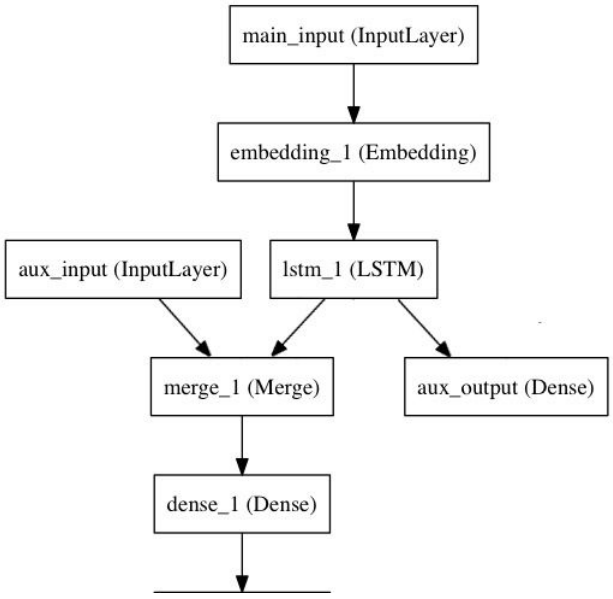
Word2Vec, google's pre-trained embedding model, is used on the tweets dataset. The result is a $[20000 \times 300]$ embedding matrix where 20000 is the number of unique words in the dataset and 300 is the dimension of the embedding.

```
[[ 0.00000000e+00  0.00000000e+00  0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [-2.25585938e-01 -1.95312500e-02  9.08203125e-02 ...  2.81982422e-02
  -1.77734375e-01 -6.04248047e-03]
 [-1.21582031e-01 -1.70898438e-01 -4.16015625e-01 ...  2.42187500e-01
   2.89062500e-01  2.06298828e-02]
 ...
 [-2.23388672e-02 -1.74560547e-02  5.71289062e-02 ... -2.31933594e-02
   4.42504883e-04  9.86328125e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 2.59765625e-01  1.18652344e-01  1.08886719e-01 ...  2.30468750e-01
   2.50000000e-01  5.58593750e-01]]
```

Building Model

The functional API makes it easy to manipulate a large number of intertwined datastreams. Let's consider the following model. We seek to predict whether a tweet is depressive or not. The main input to the model will be the tweetitself, as a sequence of words, but to spice things up, our model will also have an auxiliary input, receiving extra data such as the np of likes, replies and retweets received on the tweet. The model will also be supervised via two loss functions. Using the main loss function earlier in a model is a good regularization mechanism for deep models.

Here's what our model looks like:



The Keras functional API is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

Here we insert the auxiliary loss, allowing the LSTM and Embedding layer to be trained smoothly even though the main loss will be much higher in the model.

```
auxiliary_output = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)
```

This defines a model with two inputs and two outputs:

```
model = Model(inputs=[main_input, auxiliary_input], outputs=[main_output, auxiliary_output])
```

We compile the model and assign a weight of 0.2 to the auxiliary loss. To specify different loss_weights or loss for each different output, you can use a list or a dictionary. Here we pass a single loss as the loss argument, so the same loss will be used on all outputs.

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', loss_weights=[1., 0.2])
```

We can train the model by passing it lists of input arrays and target arrays:

```
model.fit([headline_data, additional_data], [labels, labels], epochs=50, batch_size=32)
```

Inputs

Input() is used to instantiate a Keras tensor.A Keras tensor is a tensor object from the underlying backend (Theano, TensorFlow or CNTK), which we augment with certain attributes that allow us to build a Keras model just by knowing the inputs and outputs of the model.
For instance, if a, b and c are Keras tensors, it becomes possible to do:
model = Model(input=[a, b], output=c)

The added Keras attributes are:

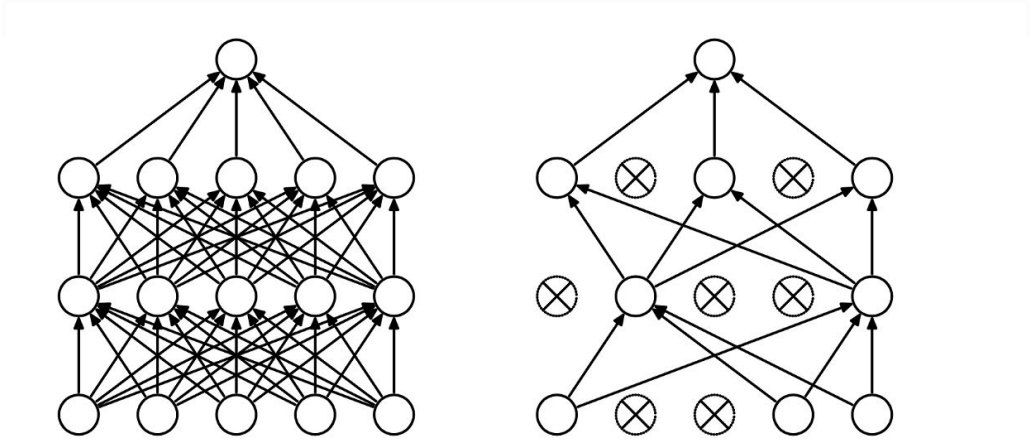
_keras_shape: Integer shape tuple propagated via Keras-side shape inference.
_keras_history: Last layer applied to the tensor. the entire layer graph is retrievable from that layer, recursively.
shape: A shape tuple (integer), not including the batch size. For instance, shape=(32,) indicates that the expected input will be batches of 32-dimensional vectors.
batch_shape: A shape tuple (integer), including the batch size. For instance, batch_shape=(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors.
dtype: The data type expected by the input, as a string (float32, float64, int32...)
sparse: A boolean specifying whether the placeholder to be created is sparse.

Dropout

Dropout is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data. It is a very efficient way of performing model averaging with neural networks. The term "dropout" refers to dropping out units in a neural network.

Dropout technique is essentially a regularization method used to prevent overfitting while training neural nets. The role of hidden units in neural networks is to approximate a ‘function’ efficiently from the available data-samples which can be generalized to unseen data. The problem is that the available data often has non-linear underlying patterns which can only be *extracted* by using a non-linear approximation function.

So, when the hidden units try to approximate a function for these samples, they tend to fit a higher order approximator (they try to create a function that accurately approximates each instance in training set) and by doing so, they overfit to the data-samples.



(a) Standard Neural Net (b) After applying dropout.

In other words, the function that has been approximated is tightly fitted to the training data and the model becomes hard to generalize or fit to test data. That means if you have a large data and you choose a neural network model with a lot layers as well as hidden units in them, then after training your neural network, you will have a very good training performance and a complex model (since a neural networks with a lot of hidden units will approximate a complex function) and in some cases, the neural network even learns the instances, which leads to good training performance but bad generalization.

If you analyze this phenomena, it serves multiple purposes. First, the hidden units in subsequent layers don't have access to *every detail* of input instance, so they might have to figure out on their own on *how to create a function for this data* efficiently by using *less knowledge of data*. It will increase the efficiency of hidden units to find patterns for unseen data. Secondly, when some of the hidden units are turned-off, it reduces the computations, so the training speed of neural nets will increase. Thirdly you will have a model that gives a *compact representation of your data* (underlying patterns) with almost same effectiveness as compared to a very complex model.

LSTM Layer

Long short-term memory (LSTM) units are units of a recurrent neural network (RNN). An RNN composed of LSTM units is often called an LSTM network. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.

LSTM networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

Concatenation

Layer that concatenates a list of inputs. It takes as input a list of tensors, all of the same shape except
for the concatenation axis, and returns a single tensor, the concatenation of all inputs.

Arguments

axis: Axis along which to concatenate.
**kwargs: standard layer keyword arguments.

Concatenating may be more natural if the two inputs aren't very closely related. However, the difference is smaller than you may think.

Note that W[x,y]=W1x+W2y, where [ ][ ] denotes concat and W is split horizontally into W1 and W2.

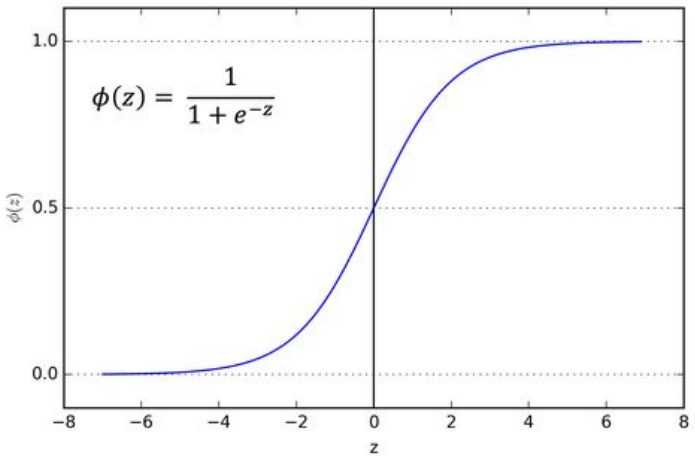
Dense Layer

We use the 'add()' function to add layers to our model. We will add dense layer as an output layer. 'Dense' is the layer type. Dense is a standard layer type that works for most cases. In a dense layer, all nodes in the previous layer connect to the nodes in the current layer. We have 10 nodes in each of our input layers. This number can also be in the hundreds or thousands. Increasing the number of nodes in each layer increases model capacity.

Activation Function

Activation functions are really important for a Artificial Neural Network to learn and make sense of something really complicated and Non-linear complex functional mappings between the inputs and response variable.They introduce non-linear properties to our Network.
'Activation' is the activation function for the layer. An activation function allows models to take into account nonlinear relationships. For example, if you are predicting diabetes in patients, going from age 10 to 11 is different than going from age 60–61. We'll be using the sigmoid function.

A sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve. Often, sigmoid function refers to the special case of the logistic function shown in the first figure and defined by the formula Special cases of the sigmoid function include the Gompertz curve and the ogee curve.The Sigmoid Function curve looks like a S-shape.



The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.

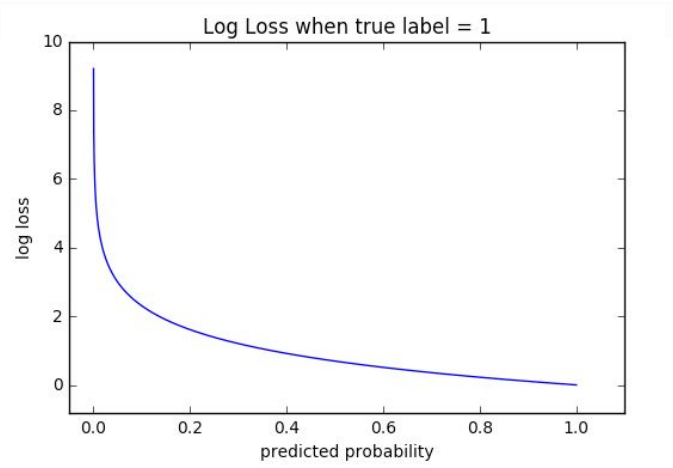
The function is differentiable.That means, we can find the slope of the sigmoid curve at any two points. The function is monotonic but function's derivative is not. The logistic sigmoid function can cause a neural network to get stuck at the training time.The softmax function is a more generalized logistic activation function which is used for multiclass classification.

Compile Model

The compilation of model takes various parameters like metrices, optimizer and loss. After building the model by adding different layers including Dense layers and Convolutional layers and using activation functions we compile the model. This model after compilation is being used to fit the dataset after splitting into training and testing data.

Loss functions

Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. So predicting a probability of .012 when the actual observation label is 1 would be bad and result in a high loss value. A perfect model would have a log loss of 0.



The graph above shows the range of possible loss values given a true observation (isDog = 1). As the predicted probability approaches 1, log loss slowly decreases. As the predicted probability decreases, however, the log loss increases rapidly. Log loss penalizes both types of errors, but especially those predictions that are confident and wrong!

Notice that when actual label is 1 (y(i) = 1), second half of function disappears whereas in case actual label is 0 (y(i) = 0) first half is dropped off. In short, we are just multiplying the log of the actual predicted probability for the ground truth class.

In binary classification, where the number of classes MM equals 2, cross-entropy can be calculated as:

$-(y\log(p)+(1-y)\log(1-p))-(\text{ylog(p)}+(1-\text{y})\log(1-\text{p}))$

Optimiser

The choice of optimization algorithm for your deep learning model can mean the difference between good results in minutes, hours, and days.The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing.It is recommended to leave the parameters of this optimizer at their default values.Nadam thus combines Adam and NAG. In order to incorporate NAG into Adam, we need to modify its momentum term mtmt.

First, let us recall the momentum update rule using our current notation :

$$\mathbf{gt=\nabla\theta tJ(\theta t)mt=\gamma mt-1+\eta gt\theta t+1=\theta t-mtgt=\nabla\theta tJ(\theta t)mt=\gamma mt-1+\eta gt\theta t+1=\theta t-mt}$$

where JJ is our objective function, γγ is the momentum decay term, and ηη is our step size. Expanding the third equation above yields:

$$\mathbf{\theta t+1=\theta t-(\gamma mt-1+\eta gt)\theta t+1=\theta t-(\gamma mt-1+\eta gt)}$$

This demonstrates again that momentum involves taking a step in the direction of the previous momentum vector and a step in the direction of the current gradient. NAG then allows us to perform a more accurate step in the gradient direction by updating the parameters with the momentum step before computing the gradient.

Metrices

A metric is a function that is used to judge the performance of your model. Metric functions are to be supplied in the metrics parameter when a model is compiled.

Example : model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['mae', 'acc'])

Metric values are recorded at the end of each epoch on the training dataset. If a validation dataset is also provided, then the metric recorded is also calculated for the validation dataset.All metrics are reported in verbose output and in the history object returned from calling the fit() function. In both cases, the name of the metric function is used as the key for the metric values. In the case of metrics for the validation dataset, the “val_” prefix is added to the key.

Both loss functions and explicitly defined Keras metrics can be used as training metrics.

Model Summary

```
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_3 (InputLayer)         (None, 140)               0
_________________________________________________________________________________________________
embedding_2 (Embedding)      (None, 140, 300)          3000000     input_3[0][0]
_________________________________________________________________________________________________
dropout_2 (Dropout)          (None, 140, 300)          0           embedding_2[0][0]
_________________________________________________________________________________________________
lstm_1 (LSTM)                (None, 300)               721200      dropout_2[0][0]
_________________________________________________________________________________________________
input_4 (InputLayer)         (None, 1)                 0
_________________________________________________________________________________________________
concatenate_1 (Concatenate)  (None, 301)               0           lstm_1[0][0]
                                                                   input_4[0][0]
_________________________________________________________________________________________________
dropout_3 (Dropout)          (None, 301)               0           concatenate_1[0][0]
_________________________________________________________________________________________________
dense_1 (Dense)              (None, 1)                 302         dropout_3[0][0]
=================================================================================================
Total params: 3,721,502
Trainable params: 721,502
Non-trainable params: 3,000,000
_________________________________________________________________________________________________
None
```

Model Training

To train, we will use the ‘fit()’ function on our model with the following five parameters:

1. Training data (train_X)
2. Target data (train_y)
3. Validation split,
4. The number of epochs and
5. Callbacks (EarlyStopping).
6. Shuffles

These parameters are explained as follows:

TRAINING & TESTING DATA : Numpy array of training data (if the model has a single input), or list of Numpy arrays (if the model has multiple inputs).

SHUFFLE : Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when steps_per_epoch is not None.

VALIDATION SPLIT : The validation split will randomly split the data into use for training and testing. During training, we will be able to see the validation loss, which give the mean squared error of our model on the validation set. We will set the validation split at 0.2, which means that 20% of the training data we provide in the model will be set aside for testing model performance.

EPOCHS : The number of epochs is the number of times the model will cycle through the data. The more epochs we run, the more the model will improve, up to a certain point. After that point, the model will stop improving during each epoch. In addition, the more epochs, the longer the model will take to run. To monitor this, we will use ‘early stopping’.

EARLY STOPPINGS : Early stopping will stop the model from training before the number of epochs is reached if the model stops improving. We will set our early stopping monitor to 3. This means that after 3 epochs in a row in which the model doesn’t improve, training will stop. Sometimes, the validation loss can stop improving then improve in the next epoch, but after 3 epochs in which the validation loss doesn’t improve, it usually won’t improve again.

Training Example Snippet :

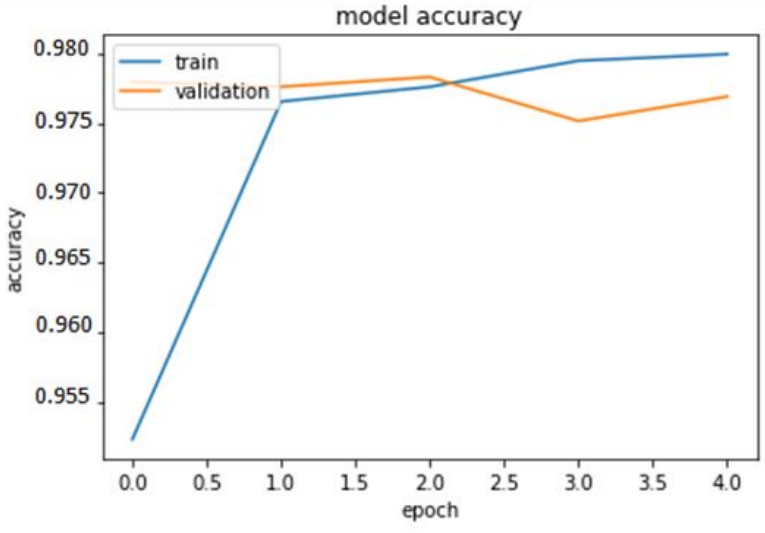
Train on 18000 samples, validate on 6000 samples
Epoch 1/10
18000/18000 [==============================] - 167s 9ms/step - loss: 0.0412 - acc: 0.9868 - val_loss: 0.0788 - val_acc: 0.9810
Epoch 2/10
18000/18000 [==============================] - 167s 9ms/step - loss: 0.0353 - acc: 0.9889 - val_loss: 0.0773 - val_acc: 0.9802
Epoch 3/10
18000/18000 [==============================] - 173s 10ms/step - loss: 0.0315 - acc: 0.9901 - val_loss: 0.0852 - val_acc: 0.9795
Epoch 4/10
18000/18000 [==============================] - 168s 9ms/step - loss: 0.0279 - acc: 0.9912 - val_loss: 0.1031 - val_acc: 0.9803
Epoch 5/10
18000/18000 [==============================] - 167s 9ms/step - loss: 0.0539 - acc: 0.9847 - val_loss: 0.0854 - val_acc: 0.9783

Result

Out of 30000 tweets, 6000 tweets are used for prediction and the accuracy of the model is given by the following formulae :

Accuracy = (Correctly guessed tweets/Total no of test_tweets) * 100

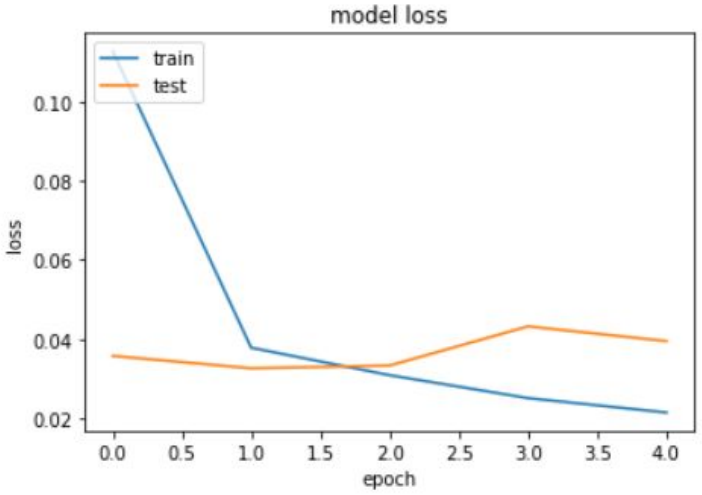
The model predicts whether a tweet is depressed or not with 97.8% accuracy. The accuracy is observed to be increasing with every epoch.



The precision, recall and f1-score are :

	precision	recall	f1-score	support
0	0.99	0.99	0.99	2400
1	0.97	0.96	0.96	640
micro avg	0.99	0.99	0.99	3040
macro avg	0.98	0.98	0.98	3040
weighted avg	0.99	0.99	0.99	3040

The loss for every epoch is plotted against epoch number and it is observed to be decreasing with every epoch.



Conclusions & Future Work

We have demonstrated the potential of using Twitter as a tool for measuring and predicting major depression in individuals. First we used Twint to collect data from twitter then we constructed well-labeled depression and non-depression datasets on Twitter. Second, using a pre-trained model of word embeddings, we analyzed the context of the words in the dataset and project the depression inclination of the tweet. Third, a layered functional model was built using the depression inclination and other features extracted from the tweet like number of likes, replies and retweets. The model consists of embedding layer, LSTM layer and dense layer. It gives a 97.8% accuracy score.

To sum up, the contributions of this project are:
(1) From the aspect of methodology, deep learning technique like LSTM is expanded to depression area.
(2) Transfer learning is used to understand the contextual meaning of data and calculate depression inclination.
(3) An association model is established between features abstracted from Twitter and depression inclination
(4) The model is used to predict whether a tweet is from depressed person or not.

In future work, we plan to expand our model from classification of tweet to classification of user as depressed or not depressed. The user profile and level of activity on social media can be used to identify potential depressed users.. Apart from the textual data, images shared can be used to detect depression in the users.

References

[1] *Afsanch Doryab, Jun Ki Min*

- Detection of Behaviour Change in people with Depression

[2] *MunMun De Chaudhary, Michael Gamon*

- Predicting Depression via Social Media

[3] *Zunaira Jamil, Diana Inkpen, Prasadith Buddhitha*

- Monitoring Tweets for Depression to Detect At-risk Users

[4] *Klaus Greff, Rupesh K. Srivastava, Jan Koutn´ık, Bas R. Steunebrink, Jurgen Schmidhuber*

- LSTM: A Search Space Odyssey

[5] *https://keras.io/*

- Keras Documentation